Um Método para Identificação de *Bad Smells* em Software a partir de Modelos UML

Henrique Gomes Nunes [#1], Mariza A. S. Bigonha [*2], Kecia A. M. Ferreira [#3], Flávio Airjan Madureira [#4]

[#] *Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)*

[*] *Departamento de Computação - Centro Federal de Educação Tecnológica de Minas Gerais (CEFET-MG)*

([1] henrique.mg.bh, [4] airjanmadureira)@gmail.com

[2] mariza@dcc.ufmg.br

[3] kecia@decom.cefetmg.br

***Resumo*—Métricas de software podem auxiliar na identificação de desvios de projeto, conhecidos na literatura como *bad smells*, e são úteis para avaliar a qualidade do código-fonte. Elas também podem ser usadas para identificar problemas estruturais nas fases iniciais do ciclo de vida do software. Este trabalho propõe um método para a identificação de *bad smells*, via métricas de software e seus respectivos valores referência, em sistemas orientados por objetos a partir de modelos UML. Neste trabalho, o método proposto foi avaliado em dois experimentos: um com o objetivo de analisar os resultados do método aplicado a versões antigas e versões refatoradas de um conjunto de seis softwares abertos; e outro com o objetivo de comparar os resultados do método com a análise manual. Os resultados desses experimentos indicam que o método proposto mostra-se útil para a identificação dos *bad smells* considerados neste trabalho.**

***Abstract*—Software metrics may aid to identify design deviances, known in the literature as bad smells and are useful for evaluating the quality of source code. They also can be used for identifying design deviances in the early stages of the software lifecycle. This work aims to contribute in this aspect, proposing a method and a tool for identifying bad smells, using software metrics, in UML models. In this work, we carried out two experiments to evaluate the proposed method: the first one aimed to evaluate the results of our method when applied to old versions as well as to refactored versions of six open source projects; in the second experiment, we compare the results of our method with the results of manual inspections. The results of these experiments indicate that our method is able to identify the bad smells analyzed in this work.**

I. Introdução

As métricas de software podem auxiliar na identificação dos desvios de projeto, conhecidos na literatura como *bad smells* [1]. Alguns trabalhos têm sido desenvolvidos para identificar *bad smells* em software com o uso de métricas a partir de código fonte [2], [3]. Todavia, poucos trabalhos propuseram avaliar *bad smells* nas fases iniciais de projetos de software [4].

Yi et al. [5] citam em seu trabalho que diversos pesquisadores definiram métricas para softwares orientados por objetos, e que um grupo específico de estudiosos se concentrou na definição de métricas para modelos da UML. Os autores ainda afirmam que não há um consenso sobre quais as métricas a serem aplicadas nesses modelos. Alguns atribuem maior importância às métricas baseadas em métodos e atributos, enquanto outros acreditam que as classes e seus relacionamentos são os mais importantes.

Assim como no caso do código fonte, as métricas aplicáveis aos diagramas da UML permitem analisar aspectos como confiabilidade, manutenibilidade e complexidade de sistemas, porém, neste caso, logo nas fases iniciais do ciclo de vida do software. Dentre os diagramas definidos na UML destaca-se o diagrama de classes, que representa classes de objetos e suas relações. A análise desse diagrama pode fornecer diversas métricas relacionadas à manutenibilidade e complexidade do software logo no início do projeto. Porém, um projeto real necessita de uma ferramenta para automatizar este processo, pois é inviável calcular tais métricas manualmente [6].

A identificação de desvios de projeto nas fases iniciais do projeto de software, como na fase de modelagem por exemplo, contribuem para que os problemas nas fases posteriores do desenvolvimento de software sejam reduzidos. O presente trabalho visa contribuir para a solução desses problemas e tem por objetivos:

- definir um método de identificação de *bad smells* em softwares a partir de diagrama de classes, aplicando métricas de software;
- desenvolver uma ferramenta que permita automatizar a coleta de métricas e a aplicação delas na identificação dos *bad smells* a partir de diagramas de classes;
- avaliar o método e a ferramenta propostos por meio de experimentos com softwares abertos.

II. Referencial Teórico

Esta seção apresenta os conceitos fundamentais relacionados a este trabalho, como métricas, valores referência e *bad smells*.

A. Métricas de Software

Sommerville [7] define medição como um valor numérico relacionado a algum atributo de software, permitindo a comparação entre atributos da mesma natureza. Ele argumenta que a medição permite realizar previsões gerais em um sistema e identificar componentes com anomalias. É possível também, por exemplo, via atributos internos de um software, definir atributos externos como facilidade de manutenção, confiabilidade, portabilidade e facilidade de uso.

Genero et al. [8] relatam o resultado de uma pesquisa, baseada em 12 outras obras, na qual apresentam conjuntos de

métricas para diagramas de classes. De acordo com Genero et al. [8], a escolha pelo modelo de diagrama de classes justifica-se pelo fato desse representar a *espinha dorsal* do desenvolvimento orientado por objetos, fornecendo informações sobre a concepção e implementação de um software logo no início de um projeto, tal que o impacto de falhas nas fases posteriores seja minimizado. Eles também afirmam que a coleta de métricas relacionadas a diagramas de classes ainda é escassa, e que por muitos anos o foco principal de medição tem sido o código fonte. Para esse trabalho foram utilizadas as seguintes métricas sugeridas por Genero et al. [8]: Número de Conexões Aferentes (NCA), Número de Métodos Públicos (NMP) e Número de Atributos Públicos (NAP).

B. Bad Smells

Para Fowler [1], *bad smell* é um indicador de um possível problema estrutural em código-fonte, que pode ser melhorado via refatoração. Fowler define o termo *bad smells* e apresenta uma lista com *bad smells* que aborda refatoração em código fonte, mas não usa as estratégias de detecção de *bad smells* para reconhecê-los. Fowler cita as características de como identificar as anomalias em código fonte, mas deixa em aberto quais metodologias poderiam ser utilizadas para tal tarefa. A seguir, são apresentadas as definições dos *bad smells* utilizados neste trabalho:

- *God Class*: corresponde a classes com alta responsabilidade no sistema. Esse *bad smell* está associado às métricas que permitem avaliar se uma classe possui muitos métodos e se muitas classes dependem da classe avaliada.
- *Indecent Exposure*: ocorre quando quando uma classe está mal encapsulada. Esse *bad smell* está associado às métricas de atributos públicos que permitem avaliar o encapsulamento das classes.
- *Shotgun Surgery*: refere-se a classes que ao serem alteradas causam impacto em outras classes. Este *bad smell* está associado às métricas de relacionamentos do tipo aferentes que permitem avaliar se a alteração em uma classe implicará em alterações em outras classes.

C. Valores Referência de Métricas de Software

Para que as métricas possam identificar os *bad smells* é necessário identificar algo que os relacione. Esse relacionamento é possível via valores referência, que definem valores numéricos para determinar quantitativamente se o valor de uma métrica é aceitável.

As métricas NCA, NMP e NAP permitem identificar os *bad smells God Class*, *Shotgun Surgery* e *Indecent Exposure*. Ferreira et al. [9] definiram valores referência para estas métricas e os classificaram como:

- *Bom*: *são os valores mais frequentes para métricas em softwares de boa qualidade*;
- *Regular*: *corresponde a valores pouco frequentes para métricas em softwares de boa qualidade*;
- *Ruim*: *corresponde a valores raros para métricas em softwares de boa qualidade*.

Métrica	*Bom*	*Regular*	*Ruim*
COF	até 0.02	0.02 a 0.14	superior a 0.14
NCA	até 1	2 a 20	superior a 20
NAP	0	1 a 10	superior a 10
NMP	até 10	11 a 40	superior a 40
DIT	média igual a 2		
LCOM	0	1 a 20	superior a 20

Tabela I. VALORES REFERÊNCIA DEFINIDOS POR FERREIRA ET AL. [9].

A idéia básica por trás das faixas *bom*, *regular* ou *ruim* é a seguinte: uma vez que os valores são frequentes em softwares, isso indica que eles correspondem à prática comum no desenvolvimento de software de alta qualidade, o que serve como um parâmetro de comparação de um software com os demais. Da mesma forma, os valores pouco frequentes indicam situações não usuais na prática, portanto, pontos a serem considerados como críticos.

Ferreira et al. [9] identificam valores referência para 6 métricas de software orientado por objetos: NCA (número de conexões aferentes), NAP (número de atributos públicos), NMP (número de métodos públicos), DIT (*depth in inheritance tree*), LCOM (*lack of cohesion in methods*) e COF (fator de acoplamento). Ferreira et al. concluem que essas métricas, exceto DIT, seguem uma distribuição de cauda pesada (*heavy-tailed distribution*), o que significa que não há um valor típico para essas métricas. Esse estudo foi realizado em um conjunto de 40 softwares abertos, de diferentes contextos e tamanhos. Os valores referência identificados estão representados na Tabela I.

D. Trabalhos Relacionados

Nos trabalhos de Girgis et al. [6], Soliman et al. [10] e Nuthakki et al. [11], os autores discutem acerca de extração de métricas a partir de modelos UML. Esses trabalhos indicam que o diagrama de classes é o modelo predominante quando o foco é medição de software a partir de modelos UML. Esses trabalhos também utilizam o formato de arquivo XMI para representar as métricas coletadas do diagrama de classes. Porém, nenhum desses três trabalhos relaciona as métricas com atributos externos de um sofware, ou seja, não fornece informações relevantes acerca do projeto a ser desenvolvido.

Os trabalhos de Marinescu [2] e [12] apresentam uma solução para transformar métricas em informações relevantes para detectar *bad smells*. Porém, esses trabalhos lidam apenas com informações coletadas em códigos fonte, que em geral passam a existir apenas em fases já avançadas do ciclo de vida dos softwares.

O trabalho de Bertrán [4] é o mais próximo do trabalho apresentado neste artigo. Nele, a autora associa métricas extraídas de diagramas de classes a *bad smells*. Para automatizar tal tarefa, a autora desenvolveu uma ferramenta para coleta desses dados. Muito embora o trabalho de Bertrán[4] tenha definido estratégias de detecção que podem ser aplicadas a diagramas UML, algumas questões foram deixadas em aberto: (1) Bertrán [4] relata, por exemplo, a importância de redefinir os valores referência utilizados em seus estudos experimentais, definidos por Marinescu [2], pois considera que a técnica utilizada por Marinescu [2] não seja a melhor forma de definí-los; (2) Bertrán [4] relata também que os experimentos foram realizados apenas por uma pessoa e que o contexto

e tamanho dos softwares utilizados não diferem tanto; (3) a autora ainda sugere que novos experimentos sejam realizados para identificar novos *bad smells* em outros diagramas da UML.

O presente trabalho visa contribuir para a solução do problema de identificar desvios de projeto em fases iniciais do ciclo de vida do sistemas. Especificamente, este trabalho de pesquisa se concentrou em definir um método para identificação de *bad smells* a partir de diagramas de classes, bem como no desenvolvimento de uma ferramenta para viabilizar a aplicação e a avaliação do método proposto. Neste sentido, as principais contribuições deste trabalho de pesquisa são:

- a definição de um método para detecção de *bad smells* em diagramas UML que utilize métricas de software e seus respectivos valores referência propostos na literatura;
- o projeto e a implantação de uma ferramenta, *UMLsmell*, que identifica *bad smells* em diagrama de classes;
- definição de uma técnica para avaliar quantitativa e qualitativamente os resultados do método proposto nesse trabalho que não dependa de código fonte;
- avaliação de uma quantidade maior de softwares e com tamanhos também maiores do que os utilizados por Marinescu [2] e Bertran [4], para avaliar a escalabilidade do método proposto neste artigo;
- realização de experimentos manuais com critérios bem definidos e com uma quantidade considerável de avaliadores.

III. Método Proposto

O objetivo deste trabalho é a automatização na identificação de *bad smells* em diagramas de classe da UML. Para atingir este objetivo foi proposto um método baseado em métricas e seus valores referências, bem como foi projetada e implementada uma ferramenta denominada *UMLsmell* que permite a aplicação do método proposto. Inicialmente, foram identificados, dentre os *bad smells* descritos na literatura, aqueles que podem ser aplicados a modelos de classe da UML. Para realização dos experimentos, foram selecionados três deles: *God Class*, *Indecent Exposure* e *Shotgun Surgery*.

A escolha desses *bad smells* foi limitada pela carência de valores referência propostos na literatura. Os valores referência usados foram definidos por Ferreira et al. [9]. A escolha desses valores referência se deu por duas razões principais: primeiro por que os mesmos consideram a frequência de vários softwares e não simplesmente a média, e segundo por que os valores referência foram avaliados empiricamente, e há poucos valores referência propostos na literatura. Dentre as métricas para as quais a autora propõe valores referência, três delas podem ser obtidas via diagramas de classes: Número de Conexões Aferentes (NCA), Número de Métodos Públicos (NMP) e Número de Atributos Públicos (NAP).

Neste trabalho, optou-se por considerar as faixas *regular* e *ruim* dos valores referência propostos por Ferreira et al. [9] na identificação de *bad smells* porque tais faixas correspondem situações não usuais na prática de desenvolvimento de software. A seguir, são descritas as estratégias de detecção propostas para esses três *bad smells*.

As métricas que definem a estratégia de detecção do *bad smell God Class* consideram seus relacionamentos com outras classes e o tamanho da classe avaliada. O *God Class* é identificado via métricas NCA e NMP. A métrica NCA verifica a influência da classe avaliada sobre as outras classes do software, ou seja, a quantidade de classes que usam os serviços da classe avaliada. A métrica NMP permite avaliar o tamanho de uma classe e a quantidade de serviços a oferecer pela quantidade de métodos. A estratégia de detecção adotada para identificação do *God Class* pelo método proposto é apresentada na Figura 1. Caso as duas métricas avaliadas estejam dentro da faixa *regular*, a avaliação do *God Class* é considerada como *regular* e caso uma das métricas esteja dentro da faixa *ruim* de acordo com os valores referência , o *God Class* é considerado como crítico (*ruim*), caso contrário a classe não é considerada como problemática.

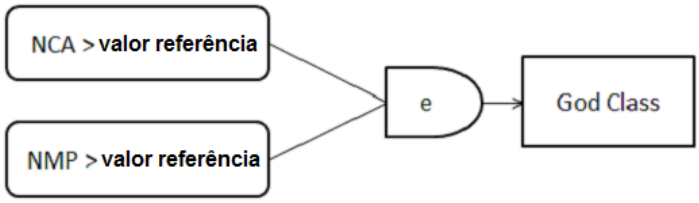

Figura 1. Estratégia de detecção para o *God Class*.

O *Shotgun Surgery* é identificado via métrica NCA, que define quantas classes dependem da classe avaliada, uma vez que quando uma classe é alterada, todas as outras classes que dependem dela estão sujeitas a mudanças. A estratégia de detecção para identificação do *Shotgun Surgery* é apresentada na Figura 2 e sua identificação em uma classe e em qual faixa se encontra, *regular* ou *ruim*, é equivalente à métrica avaliada.

Figura 2. Estratégia de detecção para o *Shotgun Surgery*.

Um bom encapsulamento acontece quando os dados, atributos, de uma classe são ocultos e seus serviços, métodos, úteis para as demais classes, são públicos [7]. Classes seguras são bem encapsuladas. O *Indecent Exposure* é um *bad smell* referente à falta de encapsulamento de classes. Ele pode ser identificado via métrica NAP, que define quantos atributos de uma classe são públicos. A estratégia de detecção para identificação do *Indecent Exposure* é apresentada na Figura 3. A identificação do *Indecent Exposure* em uma classe e em qual faixa se encontra, *regular* ou *ruim*, é equivalente à métrica avaliada.

Definidas as estratégias de detecção, o próximo passo foi definir a estrutura do *UMLsmell*. Sua estrutura é ilustrada na Figura 4.

UMLsmell recebe como entrada um arquivo XMI, no qual é realizada a análise sintática com o objetivo de extrair as

Figura 3. Estratégia de detecção para o *Indecent Exposure*.

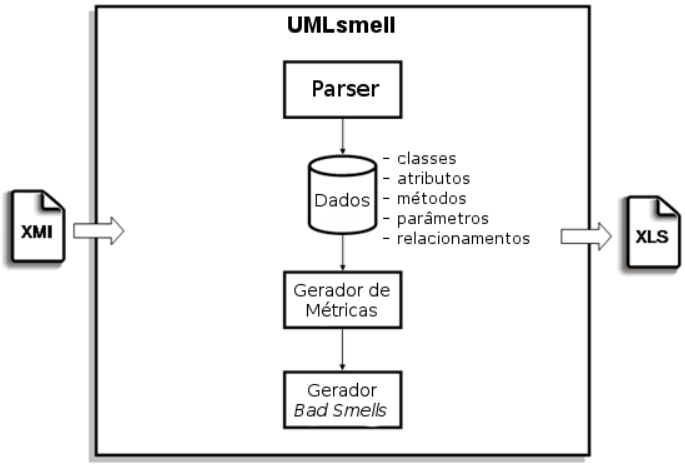

Figura 4. Estrutura da UMLsmell.

seguintes informações: classes, atributos, métodos, parâmetros e relacionamentos. Esses dados são armazenados internamente nas estruturas de dados definidas pelo programa para consultas posteriores, tal que, a partir desses dados é possível fornecer as informações de métricas das classes do software ao usuário. Essas métricas são, então, aplicadas para identificar *bad smells* conforme o método proposto neste trabalho. A ferramenta foi desenvolvida na linguagem Java e contém 3205 linhas de código.

IV. Experimentos

Esta seção relata os experimentos realizados para avaliar o método proposto e os seus respectivos resultados. O objetivo desses experimentos foi avaliar o método para identificação de *bad smells* a partir de diagramas de classes.

A. Metodologia

Os experimentos investigaram as seguintes questões de pesquisa:

RQ1: as métricas de software e seus respectivos valores referência auxiliam a identificar *bad smells* em um software?

RQ2: os *bad smells* identificados pelo método em um software são correspondentes àqueles identificados por avaliadores com formação em Engenharia de Software?

Para responder essas questões foram realizados os dois conjuntos de experimentos descritos a seguir.

- Análise de softwares reestruturados: esse experimento avalia a identificação automática de *bad smells* em duas versões de cada software, tal que uma versão é a refatoração da outra. Tendo em vista que uma refatoração é uma modificação realizada no software para melhorar sua estrutura, espera-se encontrar mais *bad smells* na versão não refatorada, indicando que o método proposto é eficaz no reconhecimento de *bad smells* automaticamente. Esse tipo de análise foi empregada por Marinescu [2] para avaliar sua proposta, porém ele a aplicou em somente um software. Neste trabalho, foram analisados os projetos de sete softwares abertos.
- Avaliação Manual: este experimento avalia manualmente a identificação de *bad smells* e os compara com os resultados da *UMLsmell* a fim de avaliar a eficácia da estratégia de detecção proposta.

B. Diagramas de Classes dos Experimentos

Foram realizados experimentos a partir dos diagramas de classes dos seguintes softwares abertos desenvolvidos em Java: *JHotdraw*, *Struts*, *HSqlDB*, *JSLP*, *Log4J* e *Sweethome 3D*. A seleção desses softwares deve-se ao fato dos mesmos serem considerados como refatorados, conforme relatado pelos autores Dig et al. [13] e Soares et al. [14]. Isso significa que os desenvolvedores do software, a cada versão lançada, se preocuparam em melhorar a arquitetura desses softwares. Para a escolha dos softwares da avaliação manual, um conjunto de softwares pequenos foram escolhidos no repositório Github e foram analisados por *UMLsmell*, para determinar quais deles possuiam mais *bad smells*. Neste processo foram escolhidos o *Picasso* e *JSLP*, pois o fato desses softwares serem pequenos viabilizou a inspeção manual pelos avaliadores.

Devido à carência de diagramas de classes para realizar os experimentos, os diagramas dos softwares utilizados para esta avaliação foram obtidos a partir de engenharia reversa usando a ferramenta Enterprise Architect [15] para ler os códigos fonte e gerar os diagramas de classes.

Como a ferramenta Enterprise Architect [15] não é capaz de criar todos os relacionamentos existentes no código fonte, utilizou-se o software Dependency Finder [16] que a partir do *bytecode* de um software é capaz de gerar um arquivo XML contendo os relacionamentos de dependência para cada classe do software. Para gerar o diagrama de classes completo automaticamente, a partir das informações obtidas na engenharia reversa do Enterprise Architect e do resultado reportado pelo Dependency Finder, neste trabalho foi desenvolvida uma ferramenta para criar automaticamente todos os relacionamentos de dependência existentes no código fonte. Essa ferramenta intermediária foi desenvolvida na linguagem Java e recebe como entrada o arquivo XMI do Enterprise Architect [15] e o arquivo XML do Dependency Finder [16]. A ferramenta realiza uma análise sintática do arquivo XML e os relacionamentos são criados no arquivo XMI da engenharia reversa feita pelo Enterprise Architect [15].

Com isso, obteve-se um arquivo XMI resultante que contém os dados de todass as classes do software e dos relacionamentos existentes entre elas.

V. Análise de Softwares Reestruturados

Esta análise buscou responder a questão de pesquisa RQ1. Para cada software avaliado, foi necessário gerar dois diagramas de classes: um deles representando a versão usada para

avaliar os *bad smells* e o outro, a versão refatorada do mesmo software. Esse detalhe foi fundamental para esse experimento, pois dado um software B que representa a refatoração de um software A, espera-se que B apresente menos *bad smells* que A, visto que a refatoração propõe melhorar os aspectos do projeto.

A Figura 5 mostra como os resultados da *UMLsmell* foram organizados para cada uma das duas versões do sistema avaliado, considerando V1 a versão do software avaliado e V2 sua versão refatorada. Para cada software usado neste experimento foram geradas três tabelas como esta, uma para cada *bad smells* avaliado. Na primeira coluna da tabela ilustrada na Figura 5 aparecem as classes presentes na Versão V1. As demais colunas, cujos conteúdos são relatados a seguir, mostram o resultado da avaliação dos *bad smells*.

	Bad Smell Avaliado				
Software Avaliado	V1	V2	Falha Real	Falso Positivo	Caso Especial
Classe 1	Regular	Regular		x	
Classe 2	Regular	Bom	x		
...	...	...	...	...	...
Classe N	Bom	Ruim			x

Figura 5. Tabela com os resultados de *UMLsmell*.

As colunas 2 e 3 contêm os resultados do *bad smell* avaliado para V1 e V2. O resultado pode ter um dos seguintes valores: *bom*, indica que os valores das métricas que avaliam o *bad smell* estão dentro da faixa *bom* de seus valores referência, ou seja, uma faixa de valores referência considerados frequentes na maioria dos softwares; *regular*, quando todas as suas métricas estão com valores referência *regulares*; *ruim*, quando ao menos uma métrica está com o valor referência *ruim*; e nos outros casos os *bad smells* estão ausentes, portanto o limiar é considerado *bom*.

A coluna denominada *Falha Real* informa se houve melhora na condição de um *bad smell* de V1 para V2, ou seja, se em relação ao *bad smell* avaliado a classe avaliada foi refatorada de V1 para V2.

A coluna denominada *Falso Positivo*, informa se de V1 para V2 uma classe não sofreu melhoria em relação ao *bad smell* avaliado, por exemplo, uma classe possuía o valor *ruim* para um *bad smell* em V1 e continuou *ruim* em V2. Como o software foi refatorado, o esperado é que o *bad smell* não esteja presente na Versão V2.

A coluna denominada *Caso Especial* apresenta o caso no qual o *bad smell* piorou de V1 para V2 em uma determinada classe. Esses casos são considerados como especiais pois a expectativa é que um software refatorado não apresente resultados piores que na versão não refatorada.

O resultado esperado nesse experimento é que a coluna *Falha Real* possua mais ocorrências do que a coluna *Caso Especial*, pois isso significa que V2 possui menos ocorrências de *bad smells* que V1, que é o que se espera de um software refatorado.

Os softwares usados nos estudos de caso foram: *JHotdraw*, *Struts*, *HSqlDB*, *JSLP*, *Log4J* e *Sweethome 3D*.

A seguir, é realizada uma análise destes resultados para verificar se as estratégias de detecção sugeridas foram eficazes na identificação automática dos *bad smells*.

A. JHotdraw

O *JHotdraw* é um *framework* para criação de editores gráficos. Neste experimento foram avaliadas as Versões 5.2 e 7.0.6 do software *JHotdraw* correspondendo a V1 e V2, respectivamente. Essa escolha se deu pelo fato de Soares et al. [14] verificarem que nesse período de tempo o software passou por diversas refatorações que provavelmente afetaram diversas classes.

Foram identificadas 171 classes na Versão V1 e 310 na Versão V2. Todas as classes de V1 mantiveram-se presentes na Versão V2, o que talvez indique que a Versão V2 tenha criado novas funcionalidades e refatorado a Versão V1 criando novas classes.

Bad Smell	*Falha Real*	*Falso Positivo*	*Caso Especial*
Shotgun Surgery	66	104	1
God Class	23	141	7
Indecent Exposure	14	127	30

Tabela II. COMPARAÇÃO DOS BAD SMELLS NO JHOTDRAW VERSÕES 5.2 E 7.0.6

A comparação entre as Versões V1 e V2 do *JHotdraw* quanto ao número de *falhas reais*, falsos positivos e *casos especiais* são apresentados na Tabela II. Esses dados permitem verificar a quantidade de refatorações de uma versão para outra, referentes aos *bad smells* avaliados. As *falhas reais* permitem quantificar quantas classes foram melhoradas de uma versão para outra, as ocorrências de falso positivo, as classes não refatoradas entre as versões, e, *casos especiais*, as pioras de uma versão para outra.

Para o *bad smell Shotgun Surgery*, de todas as 171 classes, 104 não sofreram alterações, apenas uma teve uma piora na versão posterior e 66 classes passaram por melhoria.

Em relação ao *bad smell God Class*, foi identificada uma situação similar ao *Shotgun Surgery*. Do total de 171 classes, apenas 7 pioraram em relação a esse *bad smell*, 141 se mantiveram no mesmo nível, e foram identificadas 23 classes com melhorias no *bad smell God Class*.

Para o *bad smell Indecent Exposure*, na Versão V2, as classes pioraram na avaliação em relação a esse aspecto.

B. Struts

Struts é um *framework* de desenvolvimento da camada controladora para arquitetura MVC (modelo-visão-controle) em Java. As versões avaliadas do software *Struts* foram as seguintes: V1 referindo-se à Versão 1.1 e V2, à Versão 1.2.7. Há 281 classes na versão V1 e 283 na Versão V2. Todas as classes de V1 mantiveram-se presentes na Versão V2, e apenas duas classes foram acrescentadas, portanto, é possível inferir que, se ocorreu a refatoração, talvez ela tenha ocorrido apenas internamente nas classes, ou então, houve apenas uma troca de responsabilidades entre as classes.

Bad Smell	*Falha Real*	*Falso Positivo*	*Caso Especial*
Shotgun Surgery	5	221	55
God Class	5	258	18
Indecent Exposure	6	272	3

Tabela III. COMPARAÇÃO DOS BAD SMELLS NO STRUTS VERSÕES 1.1 E 1.2.7

Os resultados de comparação entre V1 e V2 podem ser vistos na Tabela III. De acordo com os resultados, dos itens avaliados, grande parte piorou da Versão V1 para Versão V2. Os *bad smells Shotgun Surgery* e *God Class* tiveram uma quantidade muito elevada de classes que pioraram na versão posterior, enquanto a quantidade de classes que melhoraram foi muito baixa. No caso do *bad smell Indecent Exposure* foi diferente, mais classes melhoraram em vez de piorar. Porém a quantidade de melhorias e pioras é de 6 para 3, respectivamente, um valor baixo se for considerado que o *Struts* possui 271 classes. Desta forma, possivelmente as refatorações realizadas não foram no sentido de eliminar ou amenizar os *bad smells God Class* ou *Shotgun Surgery*.

Um ponto que deve ser observado diz respeito aos tipos de alterações que foram feitas pela equipe de desenvolvimento durante as refatorações. Dig et al. [13] relatam em seus estudos que a refatoração do *Struts* priorizou mover métodos, mover campos, remover métodos, mudar tipos de argumentos e renomear os métodos. Com isso, as maiores mudanças foram nos métodos e nos campos, de forma que refletiu em melhoria no aspecto de ocultação de informação, referente ao *bad smell Indecent Exposure*, mas não dos aspectos dos *bad smells God Class* e *Shotgun Surgery*.

C. HSqlDB

O *HSqlDB* é um servidor de banco de dados escrito em linguagem Java. As versões avaliadas neste experimento foram as *HSqlDB* 1.8.1, referenciadas aqui como V1, e *HSqlDB* 2.3.1, que é a versão refatorada e referenciada aqui como V2. Na Versão V1 foram identificadas 378 classes e na Versão V2, 703 classes. O aumento de classes de uma versão para outra foi de quase o dobro, o que possivelmente se deu devido à criação de novas funcionalidades, além da refatoração para melhoria das classes.

Bad Smell	*Falha Real*	*Falso Positivo*	*Caso Especial*
Shotgun Surgery	32	321	25
God Class	11	343	24
Indecent Exposure	32	314	32

Tabela IV. COMPARAÇÃO DOS BAD SMELLS NO HSQLDB VERSÕES 1.8.1 E 2.3.1

De acordo com a Tabela IV para o *bad smell Shotgun Surgery*, observa-se que houve melhoria na Versão V2 em relação a Versão V1. Por outro lado, observa-se que para o *bad smell God Class* houve uma piora. Uma possível explicação para isso é que o aumento de classes na Versão V2 pode ser resultado do acréscimo de novas funcionalidades e também do aumento da inteligência de cada classe. A preocupação em relação ao encapsulamento das classes continuou a mesma, visto que as quantidades de melhorias e pioras são as mesmas.

Soares et al. [14] afirmam que houve refatorações no software entre as versões analisadas aqui, mas não detalham os tipos de refatorações que foram realizadas. A análise dos resultados desse software mostram que, dentre os *bad smells* avaliados por *UMLsmell*, somente no aspecto de *Shotgun Surgery* houve melhora.

D. JSLP

O JSLP é uma implementação pura de SLP (*Service Location Protocol*) em Java. As versões avaliadas nos experimentos realizados foram 0.7 e 1.0, que correspondem respectivamente às Versões V1 e V2. Esse é um software pequeno, a Versão V1 possui 33 classes e V2 possui 35 classes. O fato do software ser pequeno e existir a diferença de apenas 2 classes entre as versões, indica que poucas mudanças devem ter ocorrido de uma versão para outra, o que pode resultar também em poucas alterações nos *bad smells*.

Bad Smell	*Falha Real*	*Falso Positivo*	*Caso Especial*
Shotgun Surgery	1	32	0
God Class	1	32	0
Indecent Exposure	0	0	0

Tabela V. COMPARAÇÃO DOS BAD SMELLS NO JSLP VERSÕES 0.7 E 1.0

Tanto para o *bad smell Shotgun Surgery* quanto para o *God Class* houve a melhoria de uma classe em cada um. Os resultados dessa análise mostram que de fato JSLP passou por modificações que geraram melhorias em sua estrutura nos aspectos dos *bad smells* considerados neste trabalho. Isso indica que essas melhorias foram percebidas pelo método proposto, ou seja, ele se mostrou capaz de identificar os problemas estruturais na Versão V1, bem como se mostrou sensível às melhorias realizadas na Versão V2.

E. Log4j

Log4j é uma API (*Application Programming Interface*) para fazer *log* de dados em aplicações Java. As versões avaliadas neste experimento foram as Versões 1.2 e 1.3 alpha 6, respectivamente referenciadas aqui como Versões V1 e V2. A Versão V1 possui 123 classes e a Versão V2, 215 classes. O aumento do número de classes foi significativo de uma versão para outra.

Bad Smell	*Falha Real*	*Falso Positivo*	*Caso Especial*
Shotgun Surgery	0	121	0
God Class	0	121	0
Indecent Exposure	9	106	6

Tabela VI. COMPARAÇÃO DOS BAD SMELLS NO LOG4J VERSÕES 1.2 E 1.3A6

Do ponto de vista das melhorias das classes de V1 em V2, observando a Tabela VI, percebe-se que os *bad smells Shotgun Surgery* e *God Class* não sofreram alterações entre as Versões V1 e V2, não havendo nenhuma melhora e nenhuma piora. Isso sugere que, no aspecto da distribuição da inteligência concentrada das classes e do acoplamento das mesmas, as refatorações realizadas ou as funcionalidades incluídas preservaram a qualidade estrutural do projeto do software. Em relação ao *bad smell Indecent Exposure*, nota-se mais melhorias do que pioras da Versão V1 para Versão V2.

F. Sweethome3D

O *Sweethome 3D* é uma ferramenta para projetos de interiores. As versões avaliadas nesse experimento foram as 2.5 e 3.5 referenciadas aqui como V1 e V2, respectivamente. A Versão V1 possui 160 classes, enquanto a Versão V2 possui

178 classes. Poucas classes foram acrescentadas entre as duas versões tão distantes, o que sugere que refatorações realmente ocorreram da versão V1 para V2, conforme afirmam Soares et al. [14].

Bad Smell	*Falha Real*	*Falso Positivo*	*Caso Especial*
Shotgun Surgery	83	78	0
God Class	32	124	5
Indecent Exposure	7	146	8

Tabela VII. COMPARAÇÃO DOS BAD SMELLS NO SWEETHOME 3D VERSÕES 2.5 E 3.5

Os dados reportados na Tabela VII mostram que a quantidade de *falhas reais* é alta, indicando que neste software foram realizadas refatorações que resultaram na melhoria dos *bad smells Shotgun Surgery* e *God Class*. A quantidade de classes classificadas como *caso especial* são baixas, isso porque a refatoração considerou os aspectos avaliados.

VI. AVALIAÇÃO MANUAL

Esta análise buscou responder a questão de pesquisa RQ2. Para a realização da análise manual foram escolhidos os softwares Picasso e JSLP. Picasso é uma biblioteca para o sistema operacional Android que permite baixar e armazenar imagens. Foi avaliada a Versão 1.1.1 do Picasso que possui 36 classes. JSLP é uma implementação pura de SLP (*Service Location Protocol*) em Java. Foi avaliada a Versão 1.0.0 RC5 do JSLP que possui 35 classes. A escolha desses softwares se deu pelo fato de *UMLsmell* identificar *bad smells* em algumas classes deles e também por que os softwares são de um tamanho viável para avaliação manual.

Para realizar a avaliação manual dos *bad smells*, 15 avaliadores, alunos de graduação de cursos da área de Computação, se voluntariaram. Os alunos possuem formação em Engenharia de Software. Para estes avaliadores foram disponibilizados: (1) em formato digital, os diagramas de classes dos softwares Picasso e JSLP (2) as definições dos *bad smells God Class*, *Shotgun Surgery* e *Indecent Exposure*, e, (3) para cada software foi distribuída uma lista com classes pré-selecionadas para que os avaliadores julgassem se uma classe possuia, um ou mais *bad smells*; ou se ela não possuia *bad smell*. Não foram apresentadas aos avaliadores as estratégias de detecção e as métricas de software utilizadas no método proposto para identificar os *bad smells*.

Na análise dos resultados produzidos pelos avaliadores, as classes consideradas como problemáticas foram as que em mais de 40% dos questionários os avaliadores julgaram possuir um determinado *bad smell*.

A. Resultado da Avaliação Manual da Biblioteca Picasso

Das 36 classes do Picasso, 15 foram avaliadas. Foram escolhidas todas as classes com *bad smells* e algumas sem falhas de projeto, incentivando uma análise real dos avaliadores.

A Tabela VIII mostra para cada uma das 15 classes do Sistema Picasso e para cada um dos *bad smells*: *God Class*, *Shotgun Surgery* e *Indecent Exposure*, a porcentagem de avaliadores que identificaram um determinado *bad smell* em uma dada classe. O resultado da avaliação do Picasso pela *UMLsmell* é apresentado na Tabela IX.

Classes	*God Class*	*Shotgun Surgery*	*Indecent Exposure*
Cache	20%	53%	33%
Loader	13%	67%	7%
Listener	0%	40%	13%
Picasso	100%	33%	7%
PicassoTest	80%	7%	13%
PicassoTransformTest	20%	0%	40%
RequestBuilder	33%	7%	13%
RequestBuilderTest	20%	0%	33%
StatsSnapshot	0%	0%	100%
TargetRequestTest	0%	0%	20%
TestTransformation	0%	7%	20%
UrlConnectionLoader	0%	13%	13%
ResponseCacheIcs	13%	7%	7%
UrlConnectionLoaderTest	0%	13%	13%
Utils	27%	7%	0%

Tabela VIII. AVALIAÇÃO MANUAL DO *Picasso*.

Classes	*God Class*	*Shotgun Surgery*	*Indecent Exposure*
Cache	*Bom*	*Regular*	*Bom*
Loader	*Bom*	*Regular*	*Bom*
Listener	*Bom*	*Regular*	*Bom*
Picasso	*Bom*	*Regular*	*Bom*
PicassoTest	*Ruim*	*Bom*	*Bom*
PicassoTransformTest	*Bom*	*Bom*	*Bom*
RequestBuilder	*Bom*	*Bom*	*Bom*
RequestBuilderTest	*Bom*	*Bom*	*Bom*
StatsSnapshot	*Bom*	*Bom*	*Ruim*
TargetRequestTest	*Bom*	*Bom*	*Bom*
TestTransformation	*Bom*	*Bom*	*Bom*
UrlConnectionLoader	*Bom*	*Bom*	*Bom*
ResponseCacheIcs	*Bom*	*Bom*	*Bom*
UrlConnectionLoaderTest	*Bom*	*Bom*	*Bom*
Utils	*Bom*	*Bom*	*Bom*

Tabela IX. AVALIAÇÃO DO *Picasso* PELA *UMLsmell*.

Para o *bad smell God Class* a maioria dos avaliadores julgaram que duas classes possuíam tal problema, *Picasso* e *PicassoTest*. Dessas duas classes a *UMLsmell* julgou apenas uma delas como problemática, classe *PicassoTest*. A classe *Picasso*, apesar de ser considerada como problemática pelos avaliadores, não foi identificado o *God Class* pela *UMLsmell*. Todavia, no método proposto, são consideradas duas métricas para detectar *God Class*: NMP e NCA. *UMLsmell* reportou a métrica NMP como *regular* e NCA como *bom*. Dessa forma, de acordo com a estratégia de detecção empregada no método, essa classe não foi indicada com esse *bad smell*.

Nas classes visualmente grandes devido a quantidade de atributos e métodos, mas com uma quantidade de relacionamentos aferentes muito baixo, os avaliadores identificaram *God Class*, enquanto *UMLsmell* não o identificou. Tal fato pode ter contribuído para que os resultados dos avaliadores e da *UMLsmell* fossem diferentes.

Em relação ao *bad smell Shotgun Surgery*, foram identificadas 4 classes com tal problema pela *UMLsmell*, *Cache*, *Loader*, *Listener* e *Picasso*. Essas classes foram também aquelas com maior porcentagem na identificação dos avaliadores.

O *bad smell Indecent Exposure* foi identificado por unanimidade na classe *StatsSnapshot* pelos avaliadores. A única classe identificada com *Indecent Exposure* pela *UMLsmell* também foi a classe *StatsSnapshot*. Nas outras classes, menos de 40% dos avaliadores consideraram que elas possuem *Indecent Exposure*.

De forma geral, neste experimento, os resultados de *UMLsmell* foram coincidentes com aqueles reportados pelos avaliadores.

B. Resultado da Avaliação Manual do Protocolo JSLP

Na Tabela X são apresentados os resultados da avaliação manual do JSLP. Na Tabela XI são apresentados os resultados da avaliação automática do JSLP feita por *UMLsmell*.

Classes	*God Class*	*Shotgun Surgery*	*Indecent Exposure*
AuthenticationBlock	13%	73%	27%
Filter	0%	73%	13%
ReplyMessage	0%	67%	0%
ServiceLocationException	27%	20%	60%
ServiceLocationManager	0%	73%	7%
ServiceType	7%	53%	27%
ServiceURL	40%	73%	40%
SLPConfiguration	80%	60%	53%
SLPManager	73%	67%	20%
SLPMessage	67%	60%	60%

Tabela X. AVALIAÇÃO MANUAL DO *JSLP*

Classes	*God Class*	*Shotgun Surgery*	*Indecent Exposure*
AuthenticationBlock	*Bom*	*Regular*	*Bom*
Filter	*Bom*	*Regular*	*Bom*
ReplyMessage	*Bom*	*Regular*	*Bom*
ServiceLocationException	*Bom*	*Regular*	*Ruim*
ServiceLocationManager	*Bom*	*Bom*	*Bom*
ServiceType	*Regular*	*Regular*	*Bom*
ServiceURL	*Regular*	*Regular*	*Regular*
SLPConfiguration	*Bom*	*Regular*	*Bom*
SLPManager	*Bom*	*Regular*	*Bom*
SLPMessage	*Bom*	*Regular*	*Ruim*

Tabela XI. AVALIAÇÃO DO *JSLP* PELA *UMLsmell*

Das duas classes identificadas com *God Class* nos resultados da *UMLsmell*, apenas uma foi considerada como problematica pelos avaliadores, a classe *ServiceURL*. Os avaliadores também julgaram que as três últimas classes que aparecem na Tabela X possuem *God Class*, enquanto a *UMLsmell* julgou que as mesmas não possuem tal problema. Essas três classes possuem muitos atributos e métodos, mas poucas conexões aferentes.

Das dez classes avaliadas, foram identificadas oito com o *bad smell Shotgun Surgery* tanto pelos avaliadores, quanto pela *UMLsmell*. Apenas a classe *ServiceLocationException* foi considerada como problemática pela *UMLsmell* mas não pelos avaliadores. Ocorreu outro caso em que os avaliadores consideraram a classe *ServiceLocationManager* como problemática, mas a *UMLsmell* não.

Dentre as quatro classes identificadas com o *bad smell Indecent Exposure* pelos avaliadores, três constam como problemáticas nos resultados da *UMLsmell*.

Neste experimento também, os resultados reportados por *UMLsmell* são muito próximos daqueles reportados pelos avaliadores.

VII. DISCUSSÃO DOS RESULTADOS

Na avaliação do experimento dos softwares reestruturados, em quase todos os softwares foram identificadas ao menos uma melhoria de *bad smells* de uma versão para outra refatorada. Este resultado confirma a eficácia do método proposto, visto que foram encontrados mais falhas na versão não refatorada dos softwares avaliados.

Essas melhorias ficaram evidentes nos softwares *Jhotdraw* e *Sweethome 3D*. Nesses dois softwares, a quantidade de *bad smells* foi muito superior em V1 do que em V2, o que está de acordo com o que foi relatado na literatura que diz que a Versão V2 é uma refatoração de V1. Nesses softwares foram constatados que principalmente nos *bad smells Shotgun Surgery* e *God Class* a quantidade de classes que apresentavam esses problemas diminuiu muito mais do que cresceu, apontando que a estrutura do sistema realmente passou por melhorias. No entanto, nesses três casos, não se pode dizer o mesmo em relação ao *Indecent Exposure*, que em alguns casos a quantidade de classes que apresentavam esse *bad smell* foi mantida ou aumentou da Versão V1 para V2. Tal fato pode ter ocorrido porque esse *bad smell* pode não ter sido alvo de atenções durante as refatorações.

Também foram observadas melhorias mais discretas nos softwares *Struts*, *Log4j*, *HSqlDB* e *JSLP*. Neles, nos três *bad smells* avaliados, ocorreram menos melhorias da Versão V1 para a Versão V2, sendo que em alguns casos ocorreu aumento da quantidade de *bad smells*. Mas, ainda assim, foi possível detectar que a refatoração ocorreu em alguns casos nos quais os *bad smells* reduziram, por exemplo, o *Indecent Exposure* no software *Struts*.

A avaliação manual identificou resultados muito próximos dos apresentados pela *UMLsmell*. Para os *bad smells Shotgun Surgery* e *Indecent Exposure*, as classes identificadas como problemáticas pela *UMLsmell* foram as classes com maior frequência de identificação pelos avaliadores. Das classes em que *UMLsmell* acusou a presença do *bad smell God Class* a maioria também foi identificada com esses *bad smells* pelos avaliadores. Quatro classes identificadas como problemática pelos avaliadores, não foram identificadas como problemáticas pela *UMLsmell*. Esses resultados indicam que as estratégias de detecção definidas neste estudo, estão próximas dos resultados dos avaliadores. No caso de *God Class*, houve uma diferença maior entre os resultados do método proposto e a análise manual. Uma explicação possível para isso é que os avaliadores provavelmente consideraram apenas o tamanho das classes, enquanto a estratégia de detecção do método proposto considera também a quantidade de classes que utilizam a classe avaliada.

Os resultados dos experimentos realizados em softwares tidos como refatorados mostram que em todos os softwares foram identificadas ao menos uma melhoria de *bad smells* de uma versão para outra refatorada. Esse resultado indica que o método proposto é capaz de detectar automaticamente os *bad smells* considerados neste trabalho, visto que foram encontradas mais falhas na versão não refatorada dos softwares avaliados. Com isso, conclui-se que a resposta para RQ1 é positiva. Da mesma forma os resultados da avaliação manual se mostraram muito próximos daqueles produzidos por *UMLsmell*, o que indica que a resposta para a RQ2 também é positiva.

VIII. LIMITAÇÕES

O método descrito neste trabalho considera poucos *bad smells*. Essa limitação se deve ao fato de haver poucas métricas para os quais há valores referência propostos na literatura.

As demais limitações encontradas estão relacionadas a seguir:

- Diagramas de classes usados nos experimentos: o ideal seria a utilização de diagramas de classes produzidos durante a fase de projeto do software. Todavia, é difícil obter esse tipo de diagrama em projetos abertos e, em geral, há restrições por parte das organizações em fornecer qualquer dado sobre seus softwares proprietários. Devido a isso, os diagramas foram obtidos via engenharia reversa a partir do código fonte de softwares abertos.
- Desconhecimento da arquitetura e do contexto dos softwares avaliados: em uma avaliação manual, para afirmar com precisão que as classes apontadas como problemáticas realmente possuem os *bad smells* acusados pela *UMLsmell*, seria necessário que um avaliador tivesse conhecimento das classes e seus elementos. Por exemplo, uma superclasse que utiliza o padrão de projeto *Strategy* e possui muitas subclasses não deve ser considerada como problemática, pois dependendo do contexto, tal estratégia exige muitas conexões aferentes

Apesar dessas limitações, os resultados da avaliação do método proposto indicam que ele é capaz de auxiliar, de forma automática, a identificação de *bad smells* em software orientado por objetos a partir de diagrama de classes, utilizando, para isso, métricas de software e seus respectivos valores referência.

IX. Conclusão

Neste trabalho foi proposto um método para identificar desvios de projeto de software nas fases iniciais do ciclo de vida do sistema, mais especificamente na fase de projeto. O método proposto se baseia em extrair métricas de diagramas de classes e usar os valores referência propostos na literatura para tais métricas para identificar *bad smells*.

Inicialmente, foram identificados os *bad smell* que podem ser extraídos de diagramas de classes. Depois foram identificadas as principais métricas que podem ser extraídas de diagramas de classes. Em relação às métricas, duas características foram levadas em consideração: primeiro, as métricas deveriam possuir valores referência propostos na literatura e segundo, as métricas deveriam representar melhor os *bad smells* que podem ser identificados em diagramas de classes. Os valores referência usados para relacionar as métricas aos *bad smells Shotgun Surgery*, *God Class* e *Indecent Exposure* foram definidos por Ferreira et al. [9].

Neste trabalho, também foi projetada e implementada uma ferramenta, denominada *UMLsmell*, que permite aplicar o método proposto. A ferramenta identifica automaticamente *bad smells* em diagramas de classes.

Para avaliar esse método, foram conduzidos dois grupos de experimentos. O primeiro foi realizado com o objetivo de verificar se o método proposto identifica os *bad smells* considerados neste trabalho, a partir da comparação entre versões de softwares que, de acordo com relatos na literatura, passaram por refatorações. O segundo experimento teve por objetivo comparar os resultados reportados pelo método e a avaliação manual.

A análise automática consistiu em avaliar duas versões de softwares abertos, tal que a segunda versão do software em questão era tida como a refatoração da primeira versão. Era esperado que *UMLsmell* apontasse menos problemas na segunda versão se comparada com a primeira. Os resultados dessa análise mostram que todos os softwares apresentaram algum tipo de melhoria da primeira para a segunda versão.

O segundo experimento foi realizado com um grupo de 15 avaliadores. Foram distribuídos os diagramas de classes dos softwares *Picasso* e *JSLP* para que os avaliadores julgassem quais classes possuem algum tipo de *bad smell*.

A comparação entre os resultados manuais dos avaliadores e dos automáticos gerados pela *UMLsmell* permitiu identificar que, na maioria dos casos, os resultados dos avaliadores e da *UMLsmell* foram próximos, principalmente para os *bad smells Shotgun Surgery* e *Indecent Exposure*. Isso indica que as métricas e valores referência utilizados no método definido neste trabalho permitem identificar automaticamente *bad smells* que seriam apontados de forma mais custosa e demorada de forma manual.

Os experimentos realizados evidenciam que o método proposto auxilia a identificar automaticamente a presença de *bad smells* em diagramas de classes.

A. Trabalhos Futuros

Com os resultados obtidos neste trabalho, identificam-se os seguintes trabalhos futuros:

- A identificação de valores referência para outras métricas, que será importante para viabilizar a detecção de outros *bad smells* com o método proposto.
- Realizar outros experimentos com pequenos ajustes nos valores referência utilizados com o objetivo de verificar se isso melhoraria a eficácia da detecção.
- Avaliar o uso de outras métricas para avaliar os *bad smells* considerados neste trabalho a fim de refinar as estratégias de detecção sugeridas.
- Aplicar o método em diagramas de colaboração da UML, como o diagrama de sequência, a fim de verificar se é possível identificar outros *bad smells* além daqueles considerados neste trabalho.

Referências

[1] M. Fowler, *Refactoring: improving the design of existing code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[2] R. Marinescu, in *Measurement and Quality in Object-Oriented Design*. Tese de Doutorado. Timisoara, Romênia: University of Timisoara, Oct 2002.

[3] M. Lanza, R. Marinescu, and S. Ducasse, *Object-Oriented Metrics in Practice*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[4] I. M. Bertrán, "Avaliação da qualidade de software com base em modelos uml." Dissertação da PUC-RJ. Rio de Janeiro, RJ, Brasil: Pontifícia Universidade Catolica do Rio de Janeiro, 2009.

[5] T. Yi, F. Wu, and C. Gan, "A comparison of metrics for uml class diagrams," *SIGSOFT Softw. Eng. Notes*, vol. 29, no. 5, pp. 1–6, Sep. 2004. [Online]. Available: http://doi.acm.org/10.1145/1022494.1022523

[6] M. Girgis, T. Mahmoud, and R. Nour, "Uml class diagram metrics tool," in *Computer Engineering Systems, 2009. ICCES 2009. International Conference on*, dec. 2009, pp. 423 –428.

[7] I. Sommerville, *Engenharia de Software 8a ed.* São Paulo: Pearson Addison Wesley, 2007.

[8] M. Genero, M. Piattini, and C. Calero, "A survey of metrics for uml class diagrams," *Journal of Object Technology*, vol. 4, no. 9, pp. 59–92, 2005.

[9] K. A. Ferreira, M. A. Bigonha, R. S. Bigonha, L. F. Mendes, and H. C. Almeida, "Identifying thresholds for object-oriented software metrics," *Journal of Systems and Software*, vol. 85, no. 2, pp. 244–257, 2012.

[10] T. Soliman, A. El-Swesy, and S. Ahmed, "Utilizing ck metrics suite to uml models: A case study of microarray midas software," in *Informatics and Systems (INFOS), 2010 The 7th International Conference on*, march 2010, pp. 1 –6.

[11] M. K. Nuthakki, M. Mete, C. Varol, and S. C. Suh, "Uxsom: Uml generated xml to software metrics," *SIGSOFT Softw. Eng. Notes*, vol. 36, no. 3, pp. 1–6, May 2011. [Online]. Available: http://doi.acm.org/10.1145/1968587.1968609

[12] R. Marinescu, "Detection strategies: metrics-based rules for detecting design flaws," in *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, sept. 2004, pp. 350 – 359.

[13] D. Dig and R. Johnson, "The role of refactorings in api evolution," in *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*. IEEE, 2005, pp. 389–398.

[14] G. Soares, B. Catao, C. Varjao, S. Aguiar, R. Gheyi, and T. Massoni, "Analyzing refactorings on software repositories," in *Software Engineering (SBES), 2011 25th Brazilian Symposium on*. IEEE, 2011, pp. 164–173.

[15] Enterprise-Architect, "http://www.sparxsystems.com.au/, acessado em 20/11/2013," 2013. [Online]. Available: http://www.sparxsystems.com.au/

[16] DependencyFinder, "http://depfind.sourceforge.net/, acessado em 08/12/2013," 2013. [Online]. Available: http://depfind.sourceforge.net/